DISEÑO DE UNA APLICACIÓN PARALELA DE SEGMENTACIÓN DE IMÁGENES BASADA EN CORBA

Resumen

14/07/2011

Jorge Jiménez Montero

Supervisor: David Expósito Singh

Año académico: 2010/2011

En el área de la informática, la segmentación se refiere al proceso de particionar una imagen digital en múltiples segmentos (conjuntos de píxeles, también conocido como super-píxeles). El objetivo de la segmentación es el de simplificar y/o cambiar la representación de una imagen en algo que es más significativo y más fácil de analizar. La segmentación de la imagen se suele utilizar para localizar los objetos y los límites (líneas, curvas, etc.) en las susodichas imágenes. Más precisamente, la segmentación de imágenes es el proceso de asignar una etiqueta a cada píxel de una imagen de tal manera que los píxeles con la misma etiqueta son aquellos que comparten ciertas características.

La segmentación de imágenes es un problema de mucho tiempo atrás, que presenta varias dificultades y desafíos, debido a dos razones principales:

El primer desafío es la complejidad fundamental de la partición de una imagen global como el conjunto de componentes estructurales. El objetivo de la segmentación de imágenes es procesar la imagen global mediante la detección de un conjunto de regiones que lo componen. La identificación de estas regiones depende de aspectos tales como cambios de textura, cambios de luz, curvas, líneas, etc.

Y el segundo desafío, que la problemática de segmentación de imágenes presenta es la propia ambigüedad que existe en la percepción de una imagen. Imágenes del mundo real son fundamentalmente ambiguas y nuestra percepción de una imagen cambia con el paso del tiempo.

El gran desarrollo de los ordenadores personales permite encontrar nuevas aplicaciones para el procesamiento y análisis digital de imágenes. La segmentación de imágenes digitales es una parte importante de muchas tareas en el procesamiento digital y análisis. Una amplia variedad de métodos y algoritmos han estado disponibles recientemente para hacer frente a los problemas de la segmentación de imágenes.

Esta amplia variedad de algoritmos y métodos en dicha problemática pueden ser clasificados fundamentalmente en cuatro categorías:

- Edge-based techniques.
- Region-based techniques.
- Deformable models.
- Global optimization approaches.

Para algunos de nosotros identificar y reconocer un objeto en una imagen puede ser una tarea simple. Sin embargo, la automatización de esta acción común para el ojo humano puede convertirse en una ardua tarea para un algoritmo de procesamiento de imágenes, y este es el problema de los algoritmos de segmentación de imágenes que tratan de resolver. Prueba de ello es el estado actual de la problemática, que amplia gama de posibilidades con resultados que "no son siempre satisfactorios."

Con el fin de hacer frente al problema de la aparición de resultados que "no son siempre satisfactorios", dependiendo del algoritmo utilizado, algunos investigadores desarrollaron un algoritmo usando MPI 1.2 para la comunicación entre las diferentes máquinas que intervienen en el procesamiento de cada una de las secciones de la imagen que son paralelizadas

(Pichel, Singh y Rivera, 2005). Así pues, este algoritmo divide la imagen en diversas regiones y las envía cada una de ellas a un ordenador para ser procesada de forma paralela.

El coste del procesamiento de imágenes es bastante grande, por tanto resulta obvio plantease el uso de la computación paralela para el tratamiento de dicha problemática. También hay que señalar que el uso de diferentes máquinas permite a cada usuario trabajar en una plataforma propia con un lenguaje de programación diferente, y por lo tanto no estaría sujeta a una determinada plataforma o lenguaje de programación. Por esta razón, la motivación de este proyecto es utilizar el algoritmo paralelo de segmentación en plataformas heterogéneas.

En la actualidad, el diseño de aplicaciones distribuidas es facilitado en gran medida por el diseño de sistemas basados en el uso de objetos distribuidos.

En este contexto, una de las arquitecturas distribuidas más ampliamente desarrollada es CORBA. CORBA es una arquitectura estándar para sistemas de objetos distribuidos. CORBA permite interactuar entre objetos distribuidos y heterogéneos.

De esta manera, como una mejora frente a la desventaja que ofrece MPI, CORBA ofrece interoperabilidad que permite que la aplicación distribuida sea mucho más flexible porque los programadores pueden trabajar con diferentes lenguajes de programación en diferentes plataformas.

El objetivo de este proyecto es evaluar las posibilidades de utilizar CORBA para la computación de alto rendimiento con el fin de lograr una efectiva reducción (a través de la computación paralela) en el rendimiento y la ejecución de una aplicación. Como caso de estudio, el autor propone el desarrollo de una aplicación que utiliza CORBA como protocolo de comunicación en un algoritmo de segmentación de imágenes, previamente desarrollado.

Es obvio que en este final del proyecto, no sólo se debe reemplazar el código, sino que también se debe llevar a cabo un estudio del rendimiento tras los cambios aplicados. En este estudio se utilizan diferentes imágenes, con diferentes tamaños y se procesan en una computadora local. En la evaluación de los resultados, analizamos el rendimiento que el sistema ofrece (tiempos de comunicación y los tiempos de cálculo) con MPI. Y, al final, los resultados que se han obtenido con el uso de CORBA.

El objetivo final de este proyecto final de carrera es el de examinar la idoneidad de CORBA con el algoritmo que otros investigadores habían desarrollado previamente, proporcionando un conjunto de conclusiones obtenidas tras la finalización del proyecto. Además de indicar posibles direcciones para futuras investigaciones y explorar las posibilidades de mejora o ampliación del código CORBA utilizado.

En otras palabras, el objetivo de este proyecto es llevar a cabo un estudio de viabilidad para analizar si se puede usar CORBA en lugar de MPI.

Con el objetivo de hacer frente a la programación en paralelo, el modelo de paso de mensajes se ha convertido en uno de los mejores paradigmas en la computación paralela.

Numerosos estudios detectaron el crecimiento de la computación paralela y muchos proyectos de estandarización comenzaron a aparecer antes de MPI. Es un hecho que durante este periodo de tiempo, las compañías sacaban sus propias versiones de paso de mensajes, sin seguir ningún tipo de estándar, intentando controlar el mercado. Había dos razones fundamentales para esta situación:

En primer lugar, no existía un estándar ya consolidado y tampoco se hizo ningún esfuerzo útil para crearlo.

En segundo lugar y como consecuencia de que no había ningún estándar, los vendedores consideraban que la diferencia con el resto de librerías y la falta de aspectos en común con la competencia se trataban de una ventaja competitiva y se centraban en seguir desarrollando sus propias librerías, en definitiva todo esto implicaba que no eran portables entre las diferentes implementaciones.

En este entorno, el proceso de crear una norma para permitir la portabilidad de los códigos de paso de mensajes de aplicaciones comenzó en un foro sobre Message Passing Standardization en abril de 1992, y el foro sobre Message Passing Interface (MPI) organizó por si mismo la Conferencia de Supercomputación de 1992 (*the Supercomputing '92 Conference*). No sólo el proceso de definir un estándar comenzó en ese momento, sino también el proyecto para proporcionar una implementación portable de MPI comenzó en dicho momento. Y de esta manera tanto el estándar de códigos de aplicaciones de paso de mensajes (MPI) y una implementación de la primera versión de esta norma (MPICH 1) surgieron. La idea era proporcionar información temprana sobre decisiones que se toman por el Foro MPI y proporcionar una implementación temprana para que los usuarios pudieran experimentar con las definiciones, incluso a medida que se estaban desarrollando.
MPI es la instanciación del modelo de paso de mensajes y como tal tiene una serie de ventajas frente a otros modelos:

- ✓ **Universalidad:** este modelo se ajusta bien en procesadores separados conectados por una red de comunicación. Por lo tanto, coincide con el hardware de la mayoría de los supercomputadores paralelos de hoy.
- ✓ **Expresividad:** el modelo de paso de mensajes es considerado un modelo útil y completo para expresar algoritmos paralelos.
- ✓ **Facilidad de depuración:** una de las causas más comunes de error es la inesperada sobrescritura en memoria, provocando que la depuración no sea fácil. Aunque el modelo de paso de mensajes, que controla las referencias de memoria de forma más explícita, hace que sea más fácil localizar los errores de lectura y escritura en memoria.
- ✓ **Rendimiento:** es una de las causas más importantes por las que el modelo de paso de mensajes se mantendrá permanentemente como una alternativa de modelo en la computación paralela, porque ofrece una manera al programador de asociar explícitamente datos específicos con los procesos, permitiendo así que el compilador y el hardware de gestión de caché funcionen plenamente.

El paradigma de paso de mensajes adopta un modelo de procesos de memoria distribuida, en la que cada proceso tiene su propio espacio de direcciones local. Los procesos trabajan conjuntamente para realizar una tarea de forma independiente usando únicamente los datos locales que tienen almacenados y aquellos que pueden obtener tras el intercambio de datos con otros procesos de forma explícita mediante el envío de mensajes.

Con respecto a las características que el estándar de la interfaz del modelo de paso de mensajes ofrece, MPI incluye: operaciones de intercambio de mensajes punto a punto, acciones colectivas de recolección de datos y realización de alguna operación. Por otra parte, MPI ofrece abstracción de los procesos a dos niveles. En primer lugar, los procesos se denominan de acuerdo al rango del grupo en el que se llevó a cabo la comunicación. En segundo lugar, las topologías virtuales permiten gráficos o nomenclaturas cartesianas de los procesos. Asimismo MPI ofrece tres clases adicionales de servicios: indagaciones del contexto, información de tiempo básica para la medición del rendimiento de aplicaciones y una interfaz para la monitorización del rendimiento.

En consecuencia, MPI tiene una amplia variedad de características, pero todas estas características no fueron utilizadas en el algoritmo original que más tarde fue sustituido por la comunicación CORBA.

En definitiva las funciones de comunicación MPI que tuvieron que ser sustituidas por funciones de comunicaciones implementadas en CORBA son las siguientes:

- Note: This routine is to send the regions to the rest of processes.
-
```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm )
```

- Note: This routine is to receive the regions.
-
```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
                  int source, int tag, MPI_Comm comm,
                  MPI_Status *status )
```

- Note: This routine is to execute a collective operation. There are five calls in the algorithm.
-
```
int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count,
                        MPI_Datatype datatype, MPI_Op op,
                        MPI_Comm comm )
```

- Note: This routine is to send results to root process, it is asynchronous.
-
```
int MPI_Bsend( void *buf, int count, MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm )
```

Además había dos funciones MPI que no eran de comunicación y por este motivo no fueron remplazadas por funciones CORBA, son:

- Note: This routine is to pack.
-
```
int MPI_Pack ( void *inbuf, int incount, MPI_Datatype datatype,
```

```
                    void *outbuf, int outcount, int *position,
                    MPI_Comm comm )
```

- Note: This routine is to unpack.

```
int MPI_Unpack ( void *inbuf, int insize, int *position,
                 void *outbuf, int outcount,
                 MPI_Datatype datatype, MPI_Comm comm )
```

La arquitectura CORBA es definida para trabajar con objetos distribuidos y su paradigma básico consiste en pedir un servicio a un objeto distribuido. Para conocer el conjunto de los servicios que son ofrecidos al resto de los usuarios, es necesario definir una interfaz que se define en el IDL (Interface Definition Language). Para acceder a estos servicios, la entidad que está interesada en estos servicios debe obtener la referencia del objeto distribuido que ofrece dicho servicio.

ORB es la parte más importante de la arquitectura CORBA, ya que es el encargado de tratar la solicitud del servicio al objeto remoto, donde el cliente que solicita un servicio puede ser escrito en un lenguaje de programación diferente de la implementación del objeto. Además, debemos mencionar que este paso es transparente para el usuario.

ORB es también muy importante para mantener la transparencia de las solicitudes de servicios del cliente, porque esta parte de la arquitectura es la responsable de traducir la petición que se ha ejecutado en un lenguaje de programación en el lenguaje de programación que el objeto distribuido, que implementa el servicio, ha sido programado. Quiero decir, este servicio es el responsable de la colección heterogénea de objetos con los que CORBA permite trabajar.
Sin embargo, cada lenguaje de programación requiere de su propio ORB, dependiendo del lenguaje de programación, éste será uno u otro. En este proyecto, el lenguaje de programación que se ha utilizado es C y el entorno es GNOME. En consecuencia, la implementación de ORB para C y GNOME es ORBit.

La interfaz definida en IDL cuyas funciones han sido utilizadas a su vez por las funciones CORBA que sustituyen a las funciones de comunicación implementadas en MPI pueden ser observadas en el siguiente cuadro de texto.

```
module corbanetwork {

      typedef sequence<octet> bufferByte;
      typedef sequence<long>     bufferInt;
      typedef sequence<float> bufferFloat;
      typedef sequence<double> bufferDouble;

      interface Storage {

               long initialize (in long numNodes, in long nodeId);

               long  setByte (in long idSender,
                              in bufferByte buffer);
               bufferByte getByte (in long idSender);

               long  setInt (in long idSender, in bufferInt buffer);
               bufferInt getInt (in long idSender);

               long  setFloat (in long idSender,
                              in bufferFloat buffer);
               bufferFloat getFloat (in long idSender);

               long  setDouble (in long idSender,
                              in bufferDouble buffer);
               bufferDouble getDouble (in long idSender);

               long barrier (in long state);

               long allreduce (in long datatype,
                              in long operation);
               };
      };
```

Para implementar el algoritmo de segmentación de imágenes v1.3 con CORBA, fue necesario volver a utilizar unas funciones en CORBA que fueron desarrolladas en un proyecto final de carrera previo.

Estas nuevas funciones se definen en el archivo cuyo nombre es *'corbanetworkpfc.h'* y se implementan en el archivo cuyo nombre es *'corbanetworkpfc.c'*. Estas funciones son las que en su implementación realizan llamadas sobre las funciones IDL que fueron mostradas en el previo cuadro de texto. Las funciones CORBA que remplazan, transparentemente para el usuario, las funciones MPI son:

- `void CORBA_Init (int num_nodes, int my_rank)`

- `Sus parámetros son:`

- `num_nodes: número de nodos que han sido lanzados en la ejecución.`
- `my_rank: identificador del nodo que llama a dicha función.`

- `int CORBA_Send (int nodeId, void *buf, int count, int datatype, int dest)`

- Sus parámetros son:
  - nodeId: identificador del nodo que llama a dicha función.
  - buf: buffer donde está almacenada la información a enviar.
  - count: número de elementos que se quieren enviar.
  - datatype: tipo de datos de los elementos a enviar.
  - dest: rank del nodo destino.

- `int CORBA_Recv (int nodeId, void *buf, int count, int datatype, int dest)`

- Sus parámetros son:
  - nodeId: identificador del nodo que llama a dicha función.
  - buf: buffer donde se almacena la información recibida.
  - count: máximo número de elementos que se quieren recibir.
  - datatype: tipo de datos de los elementos a recibir.
  - dest: rank del nodo emisor.

- `int CORBA_Allreduce ( int nodeId, int numNodes, void* inbuf, void* outbuf, int count, int datatype, int op)`

- Sus parámetros son:
  - nodeId: identificador del nodo que llama a dicha función.
  - numNodes: número de nodos que participan en dicha comunicación.
  - inbuf: buffer inicial que contiene la información.
  - outbuf: buffer final donde se almacena el resultado.
  - count: número de elementos del buffer inicial.
  - datatype: tipo de datos de los elementos a enviar.
  - op: tipo de operación que se quiere ejecutar.

- `int CORBA_Barrier (int nodeId)`

- Sus parámetros son:
  - nodeId: identificador del nodo que llama a dicha función.

Al remplazar las funciones MPI de comunicación por estas funciones CORBA, el algoritmo de segmentación de imágenes v1.3 presentaba algunos inconvenientes para dicho remplazo directo, puesto que una de las funciones MPI era una función asíncrona y todas las funciones CORBA se tratan de funciones síncronas. De este modo, al intentar remplazar las funciones sin realizar ningún cambio en el algoritmo se producía un interbloqueo entre los nodos.

Para poder resolver este interbloqueo, lo que se requería era modificar el algoritmo ligeramente de forma que se utilizara un patrón de comunicación no bloqueante. Un buen patrón de comunicación no bloqueante puede ser el siguiente, donde un nodo envía datos al siguiente nodo y recibe de su nodo anterior. Una explicación para dicho patrón con 4 procesos es la siguiente:

P0 envía a P1 y recibe de P3, envía a P2 y recibe de P2, y envía a P3 y recibe de P1. P1 envía a P2 y recibe de P0, envía a P3 y recibe de P3, y envía a P0 y recibe de P2.

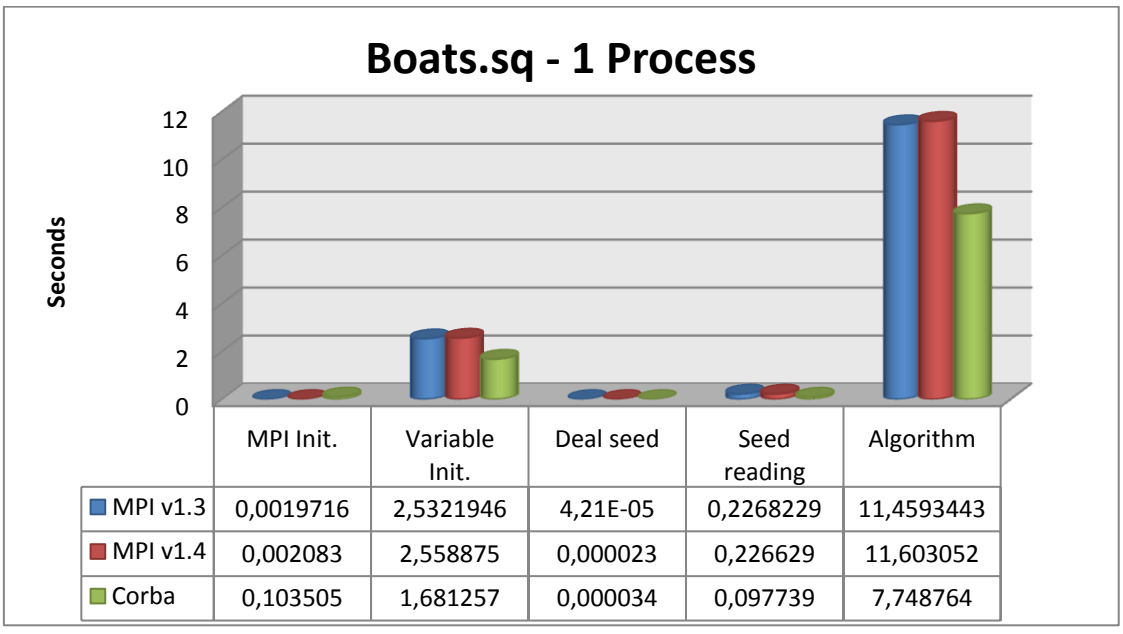
P2 envía a P3 y recibe de P1, envía a P0 y recibe de P0, y envía a P1 y recibe de P3. P3 envía a P0 y recibe de P2, envía a P1 y recibe de P1, y envía a P2 y recibe de P0.

Este patrón es un patrón regular por lo que no tiene ningún problema con interbloqueos entre nodos.

Para realizar el estudio comparativo de tiempos de computación y comunicación que nos permitían comparar la viabilidad de usar CORBA en el algoritmo de segmentación de imágenes v1.3, Tuve que llevar a cabo una importante cantidad de pruebas. De manera que los resultados que se van a ver en los gráficos son los promedios de diez pruebas diferentes. Ésto significa que para obtener un resultado aproximado de los tiempos de comunicación y computación del algoritmo con CORBA y con dos procesos, se tuvieron que realizar diez pruebas diferentes y el resultado de la media de estas diez pruebas son los que son utilizados para la elaboración de los gráficos.

Vamos a comparar los tiempos de cálculo que se han podido recoger con la versión MPI del algoritmo (las dos versiones, la original y la modificada para poder utilizar las funciones CORBA) y la versión del algoritmo con CORBA. Debo señalar que para esta ejecución se utilizó un equipo local, en el cual los gráficos más importantes son los tiempos obtenidos en la ejecución paralela con dos procesadores, porque dicho equipo local cuenta con dos núcleos y esta prueba es un resultado paralelo real. Por otro lado los tiempos de cálculo que se obtienen en el portátil con 4, 8 y 16 procesadores no son tiempos realmente paralelos porque el portátil sólo tenía dos núcleos. Por lo tanto para este resumen, únicamente voy a mostrar las gráficas elaboradas en la ejecución con 2 procesadores, puesto que es el más importante y desde este punto mostraré las conclusiones obtenidas al final del proyecto final de carrera.

Primeramente me gustaría mostrar los tiempos de ejecución y comunicación en la ejecución del algoritmo con un único proceso, que obviamente es una ejecución en secuencial, ya que no existe ningún tipo de comunicación. Esta ejecución nos servirá como punto de partida para ver la diferencia de tiempos de computación con las versiones MPI y las versiones CORBA, para después poder comparar el resto de ejecuciones paralelas.



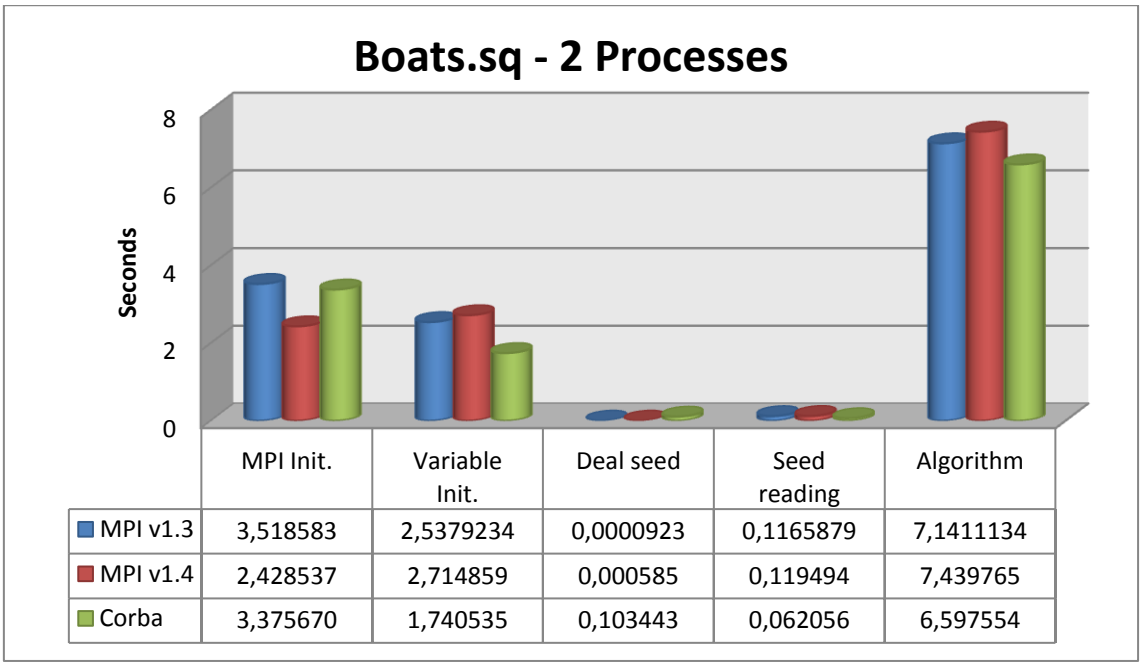
	MPI Init.	Variable Init.	Deal seed	Seed reading	Algorithm
MPI v1.3	0,0019716	2,5321946	4,21E-05	0,2268229	11,4593443
MPI v1.4	0,002083	2,558875	0,000023	0,226629	11,603052
Corba	0,103505	1,681257	0,000034	0,097739	7,748764

“Boats.sq” es una de las imágenes utilizadas en las pruebas (todas ellas son ampliamente explicadas en la memoria original del proyecto en inglés).

Como se puede observar en el gráfico, los tiempos son diferentes entre sí, aunque esta diferencia no es realmente importante. Esto se debe a que no hay ningún tipo de comunicación por lo tanto, los tiempos de cálculo entre las tres versiones del algoritmo debe ser muy similar entre sí.

Por otro lado, hay que señalar que los tiempos de CORBA son un poco mejor que en la versión de MPI. Este punto tiene que ser tomado en cuenta para las posteriores pruebas y sus respectivas comparaciones debido a que este gráfico es el punto de partida para el resto de pruebas que vienen. El aspecto importante es que la fase INIT en la versión CORBA es siempre un poco peor que en la versión MPI, incluso cuando no hay comunicación.

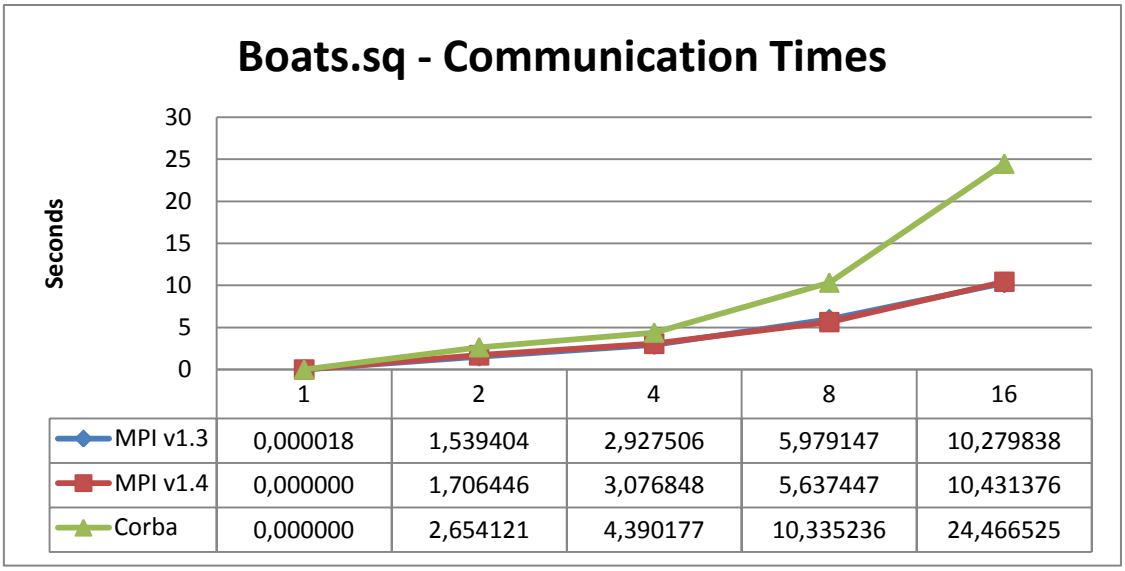
A continuación muestro los tiempos de computación obtenidos en la ejecución del algoritmo con 2 procesadores.



	MPI Init.	Variable Init.	Deal seed	Seed reading	Algorithm
MPI v1.3	3,518583	2,5379234	0,0000923	0,1165879	7,1411134
MPI v1.4	2,428537	2,714859	0,000585	0,119494	7,439765
Corba	3,375670	1,740535	0,103443	0,062056	6,597554

En este gráfico podemos ver cómo los tiempos fueron peor que en el gráfico anterior de un proceso, excepto en la fase del “Algorithm", donde el tiempo mejoró, a saber: el algoritmo es el más difícil y pesado para el equipo. Esto significa que esta fase es la razón inicial para la paralelización del algoritmo de segmentación de imágenes v1.3. Respecto al resto de las fases, como se puede ver, la versión de CORBA es peor que las versiones de MPI. Si se tiene en cuenta únicamente este gráfico no se aprecia este empeoramiento, pero en este caso debemos recordar los resultados obtenidos en la ejecución con un solo procesador y podemos observar cómo, en porcentaje, los tiempos con CORBA mejoran en menor medida que con MPI.

Para terminar con el análisis de los tiempos con la imagen cuyo nombre es Boats.sq, me gustaría mostrarle un gráfico donde podemos comparar los diferentes tiempos de comunicación que he podido recoger durante las realizadas:

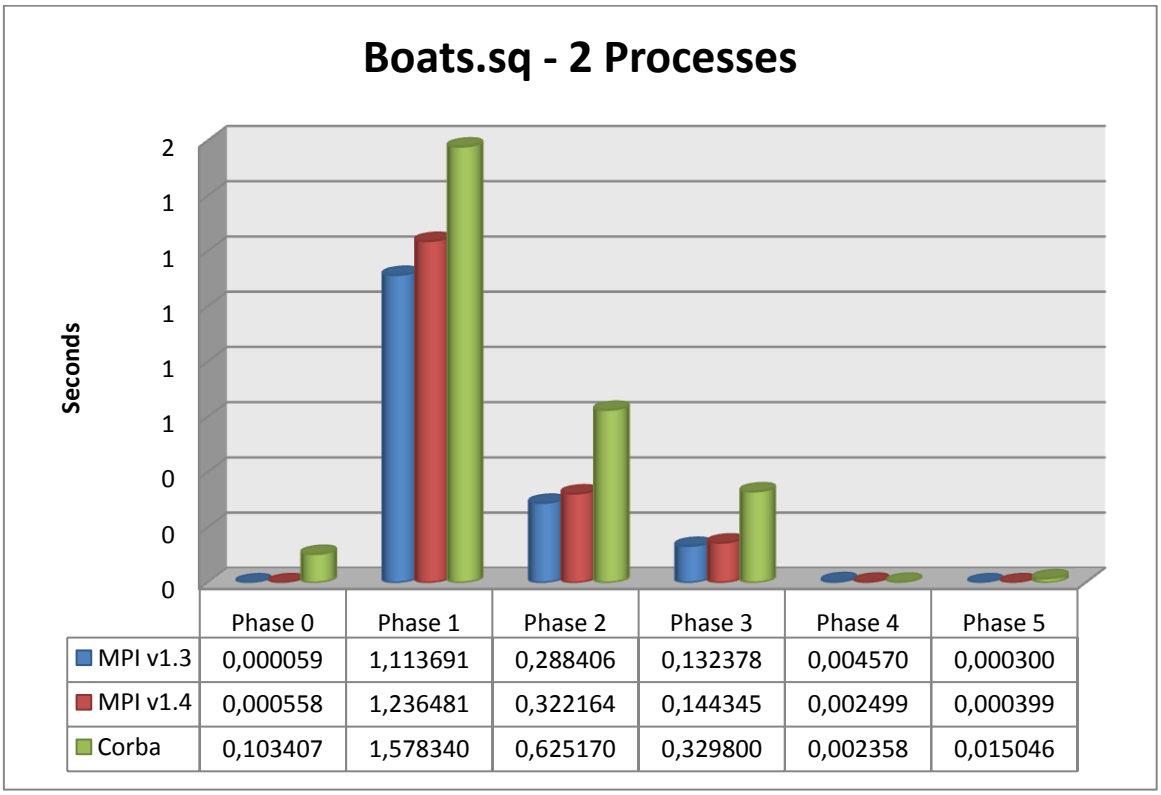


	1	2	4	8	16
MPI v1.3	0,000018	1,539404	2,927506	5,979147	10,279838
MPI v1.4	0,000000	1,706446	3,076848	5,637447	10,431376
Corba	0,000000	2,654121	4,390177	10,335236	24,466525

Obviamente, el tiempo de comunicación es 0 cuando se utiliza un único procesador en la paralelización. Y este gráfico es muy representativo porque aquí se puede ver cómo los tiempos de comunicación son más o menos lo mismo entre las versiones de MPI. Esto es así porque la implementación de la comunicación es la misma (MPI), pero el algoritmo se ha cambiado para evitar el uso de una función asíncrona.

Otro punto interesante es la comparación entre CORBA y MPI, porque como se puede ver, con CORBA los tiempos son peores en la comunicación que con MPI y ésta es la razón por la que los tiempos de computación con CORBA también son peores.

Para terminar de resumir los resultados obtenidos, me gustaría mostrarle los tiempos de comunicación que obtuve para la misma imagen en la ejecución con dos procesadores:



	Phase 0	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
MPI v1.3	0,000059	1,113691	0,288406	0,132378	0,004570	0,000300
MPI v1.4	0,000558	1,236481	0,322164	0,144345	0,002499	0,000399
Corba	0,103407	1,578340	0,625170	0,329800	0,002358	0,015046

En dicho gráfico podemos observar como la versión de CORBA siempre necesita más tiempo en la comunicación que las versiones de MPI. Obviamente, esta característica es perjudicial a los tiempos de computación, como ya he intentado resumir con la muestra de los gráficos más representativos del proyecto.

Destacar que en el proyecto se han realizado pruebas con otras imágenes obteniendo conclusiones similares a las descritas en dicho resumen.

Si analizamos los resultados obtenidos y los comparamos con los objetivos que se definieron al inicio de este informe, podemos llegar a la conclusión:

1. Que se pudo lograr el primero de los objetivos de este proyecto que consistía en reemplazar las funciones de comunicaciones MPI con las funciones de comunicaciones implementadas en CORBA. Para ello, tuvimos que utilizar el middleware que se llama Orbit 2, una implementación de la API de CORBA 2.4 en C (lenguaje de programación).
2. El segundo objetivo también se consiguió. Se trataba de analizar el rendimiento ofrecido por el algoritmo de segmentación de imágenes v1.3 que podíamos conseguir después de aplicar todos esos cambios.
3. Finalmente pudimos lograr el último objetivo que se definió al principio del proyecto final: el estudio de adaptación del algoritmo CORBA en HPC (computación de alto rendimiento). La conclusión principal de este proyecto es que la aplicación basada en CORBA no es factible debido a que introduce una sobrecarga importante en la comunicación, lo que limita el rendimiento y la escalabilidad de la aplicación paralela del algoritmo. Esta sobrecarga no es tan importante que la sobrecarga que había en el proyecto anterior, pero todavía la sobrecarga es importante.

Para terminar con el resumen, es importante indicar los posibles trabajos futuros a los que este proyecto final de carrera se presta a seguir.

Primeramente, se podría completar con otras investigaciones. Una de ellas podría ser el estudio de la ejecución de este código en un clúster con características similares a las características de Kasukabe, ofrecido por la UC3M. Aunque no se obliga de que el clúster tenga que tener las mismas características.

Además, los resultados podrían extrapolarse también a los entornos de Cloud Computing, para lo cual el coste de las comunicaciones puede ser similar al coste que se obtuvo en este proyecto.